Processor Prototyping Lab Final Report

ECE 43700

Group Members: Christopher Priebe, Victor Le

Lab Section: 5

GTA: Vaibhav Ramachandran

Due Date: May 02, 2021

1 Executive Summary/ Overview

The basis of this report is comparing our dual core pipeline processor design with our original single core processor design. To standardize our gathered data and validate the performance differences, we used a mergesort algorithm to test both dual core and pipeline processors. As far as our verification methods, we designed test benches for individual blocks such as the dcache, icache, and bus controller. System level testing was done using the automated "testasm -c" and "testasm -d" command respectively for single core and dual core. This allowed us to test our design against asm files that were given as well some we created.

Following the design of our original pipeline processor, we started to design and implement our cache block. The intention of our cache was to act as means to store memory to data and instructions that would possibly be used again for quicker access times, overall improving the speed of the program. Additionally, to improve our design we designed snooping infrastructure and a bus controller to be utilized for parallel programs. Overall, this would help our run times by increasing the effective CPI by utilizing two cores instead of one. The idea of the bus controller which is a subset of the memory controller was to react accordingly to the MSI Protocol that was defined through the data stored within a cache frame. This allowed us to finally implement our Load Linked and Store Conditional commands which achieves synchronization and allows shared memory access between the cores without compromising the resulting values.

In general, the benefit of our multicore design was improving the execution time without compromising the frequency. We achieved this by actively latching the system so that there was minimal critical path delays. The areas of performance we will be comparing include the synthesis frequency, the average instruction per clock cycle, latency, speed up, and FPGA resources.

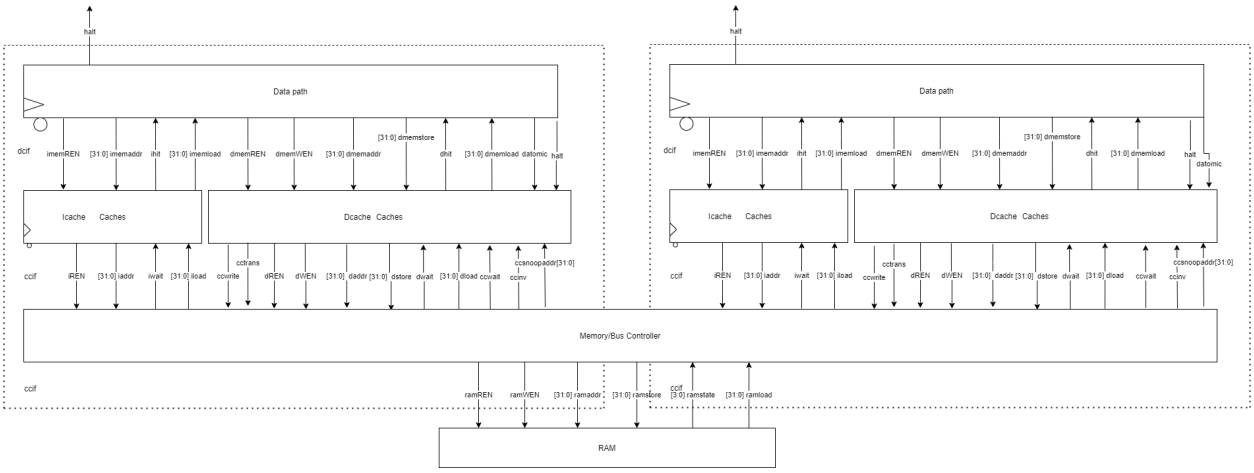


Figure 1: Multicore Block Diagram

2 Processor Design

The dual core block diagram is shown Figure 1. This diagram includes our Pipeline datapath that remains fairly unchanged besides supporting Load Linked and Store Conditional (LL/SC) instructions. The only infrastructure change for these instructions is the "datomic" signal that will allow the data cache to identify the LL/SC instruction versus a LW/SW. Additionally, there is a new caches block which contains our instruction cache and data cache. This leads to our memory controller which handles any coherence issues via the MSI-protocol bus transactions and the use of the new "cc" or cache coherence signals. Signals such as "ccwrite" (sender) and "ccinv" (receiver) are used to initiate a bus read exclusive transaction and invalidate the cache frame of the receiver. The "ccsnoopaddr" and "ccwait" signals are also used by the transaction to snoop the proper address and identify the proper MSI state. The "cctrans" and "ccwrite" signals are used for write back transaction mostly for processor write from the sender core.

The RTL block diagram for the pipelined processor is shown in Figure 2. This processor remains fairly unchanged based off our previous pipeline design, but also implements a Load Link and Store Conditional Signals. This has been added to the original 5 stage pipeline with full control hazard detection and stalling, full RAW hazard forwarding, and a branch-

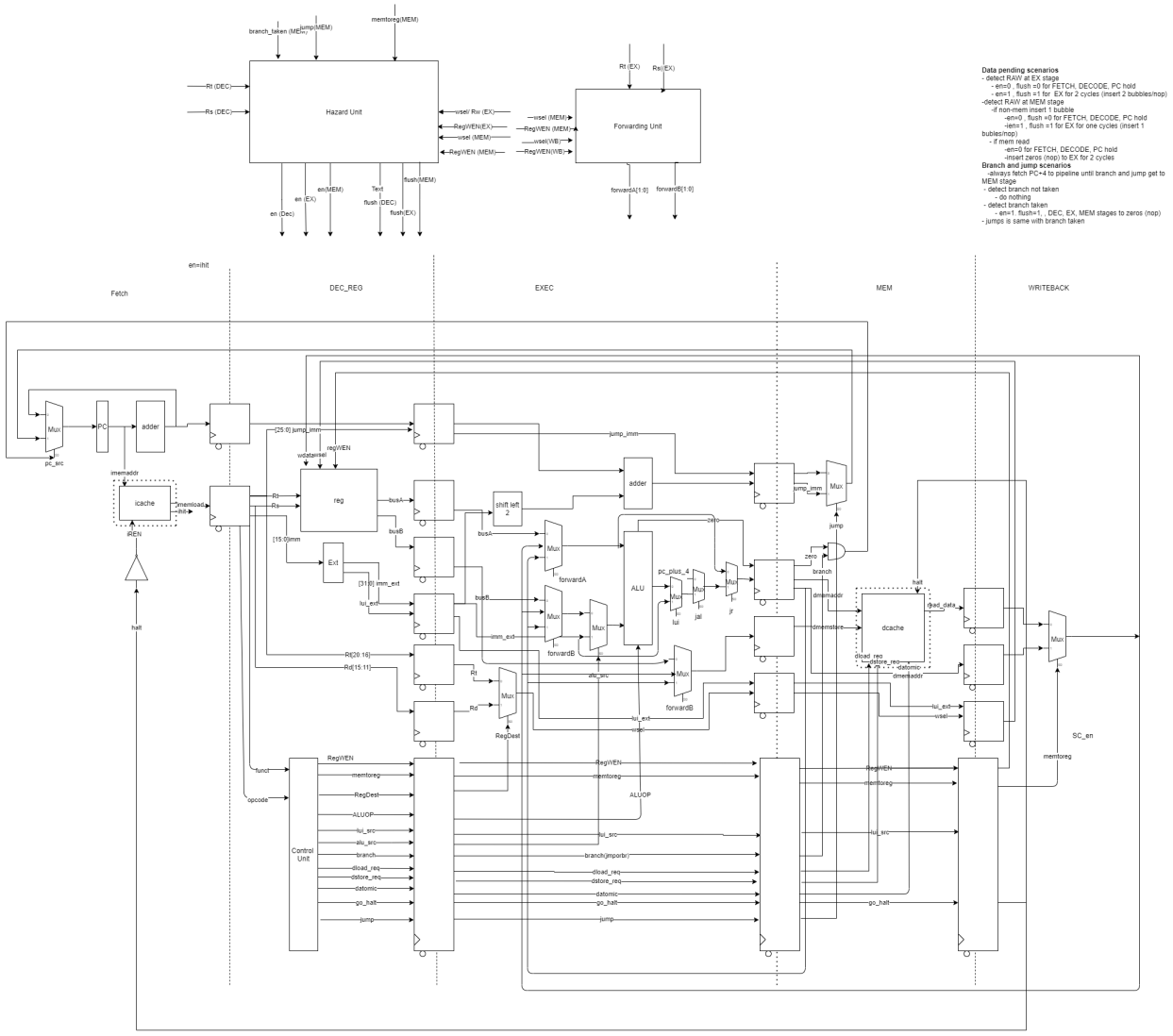


Figure 2: Pipeline Block Diagram

always-taken predictior for branch instructions.

In Figure 3 is the design for a 1 Kb data cache with 2-way associativity. These cache frames also contained the valid, dirty, tag, and 64 bits of data (2 words). The appropriate cache side to write to is determined by the LRU logic. There is also a cache controller in charge of stores, writes, and reads from cache or memory, FSM seen in Figure 4. In parallel, we also implemented a snoop side cache frame and controller for receiving transactions from

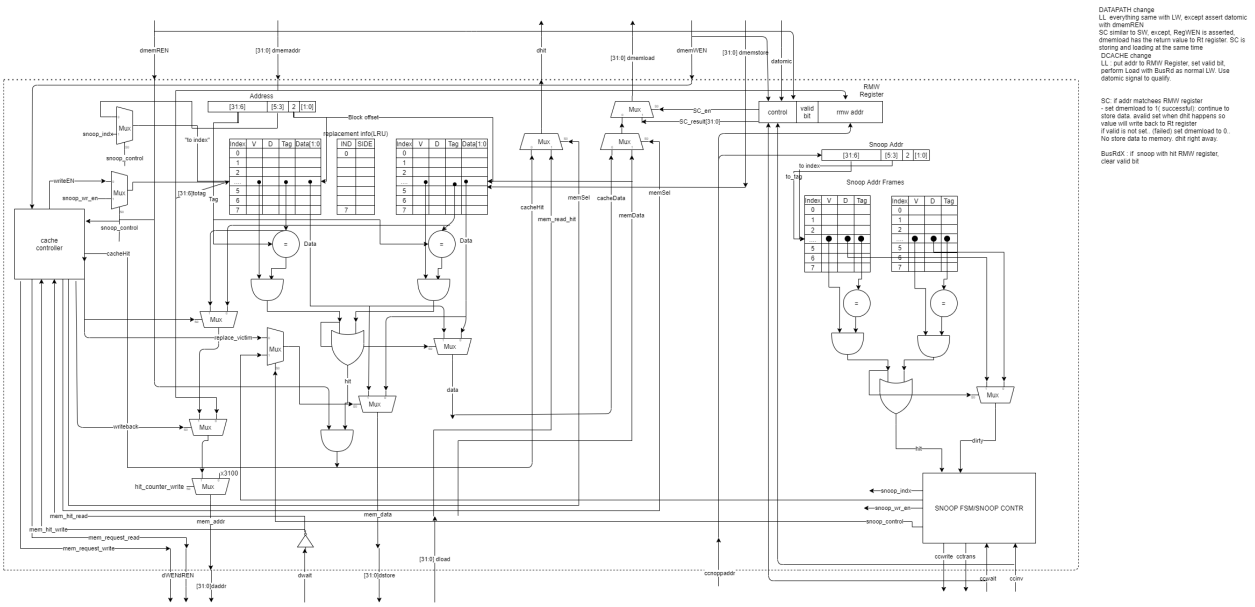


Figure 3: d-Cache Block Diagram

other cores. In Figure 5, you can see the design of our icache, which is relatively much simpler due to having only 1 word per block as well only a 1 way associative. The icache of each core are fairly independent of each other, the only interaction is the arbiter in the memory controller which give precedence to the core to retrieve instructions from next.

The Final additions to our design include Figures 6 and 7. These are the memory controller block diagram and bus controller FSM respectively. The bus controller is a subset of the memory controller and controls most of the latter besides the icache request. The bus controller as seen in Figure 7 is in charge decoding and initiating the appropriate bus tranactions based on the signals received from the dcache.

Our bus controller and dcache rely on the MSI protocol to effectively share data. The MSI protocol consist of three states including the Modify, Share, and Invalidated. These states can be determined within a cache frame, for example if the current frame is valid and dirty it would've been in the Modify state because it contained valid data but it was not in the share state as dirty would've had to been low. And that leaves invalidated being

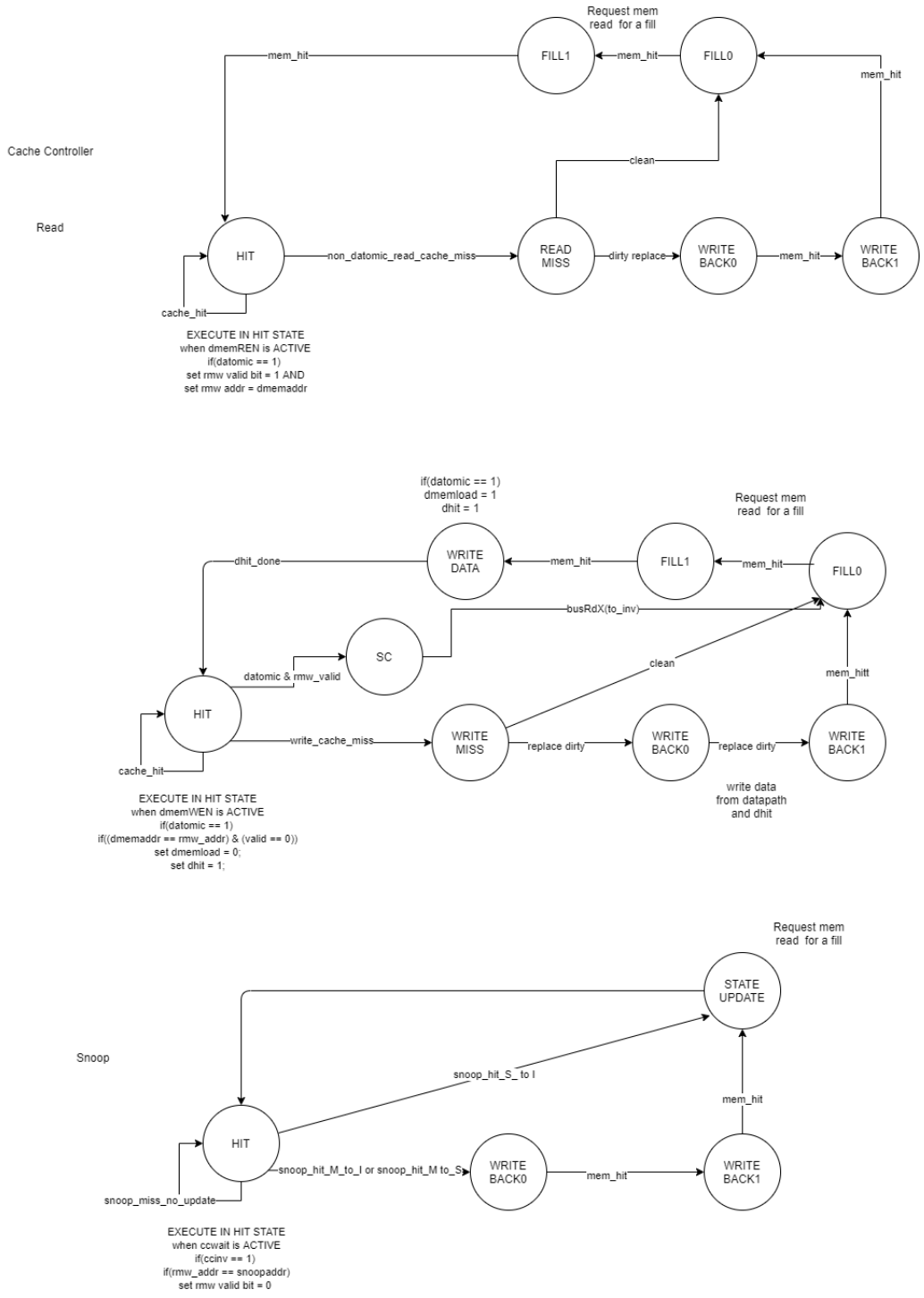
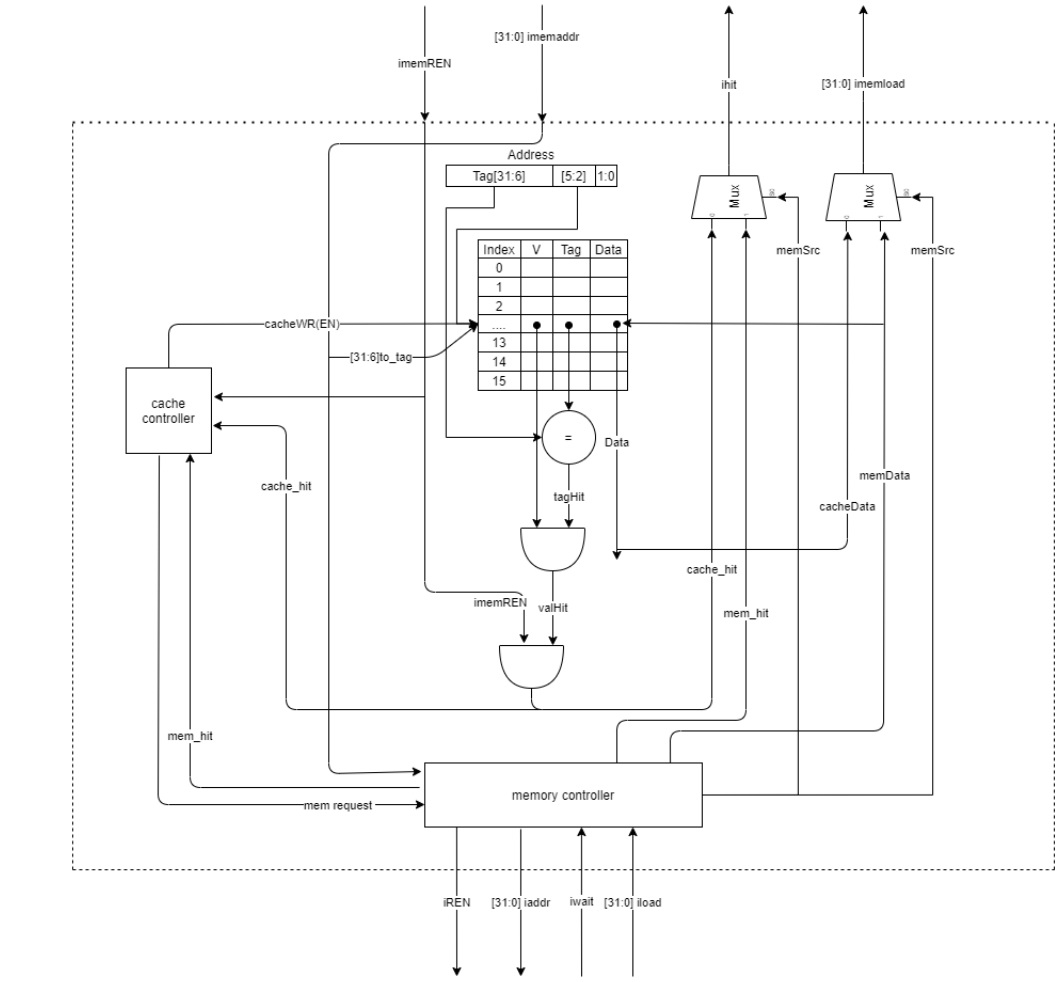


Figure 4: d-Cache FSM



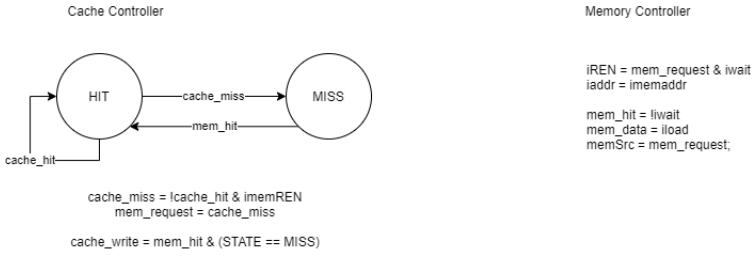
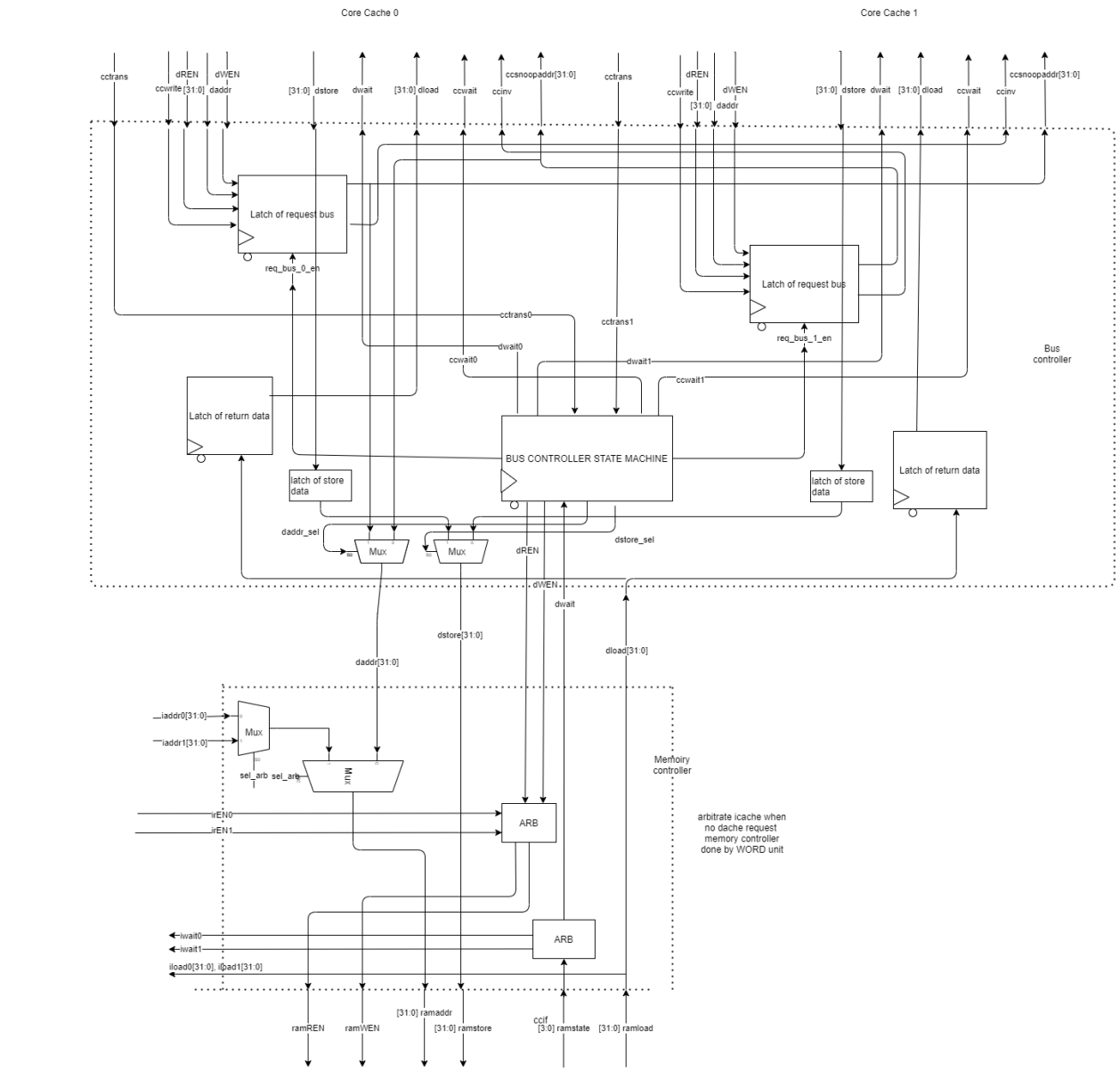


Figure 5: i-Cache Block Diagram

identified if the valid bit was low. The result of a following processor read would cause a BusRd transaction within the memory controller and both cores cache a particular address



dREN and dWEN only once per block, bus controller will generate second word request to memory controller, there are still 2 dwait of zeros per block request.
(we don't need memory controller block in dcache)

ccwrite - cache controller holds this signal if a processor writes causes this bus transaction read (so distinguish between BusRd and BusRdX

ccwait - bus controller holds this signal high when asking the cache controller for a snoop.
ccsnoopaddr - 32 bit snoop address
ccinv - bus controller holds this signal together with a ccwait to distinguish BusRd and BusRdX (ccinv is high for BusRdX
cctrans - cache controller holds this signal high until MSI transition caused by bus transaction complete

Figure 6: Memory Controller Diagram

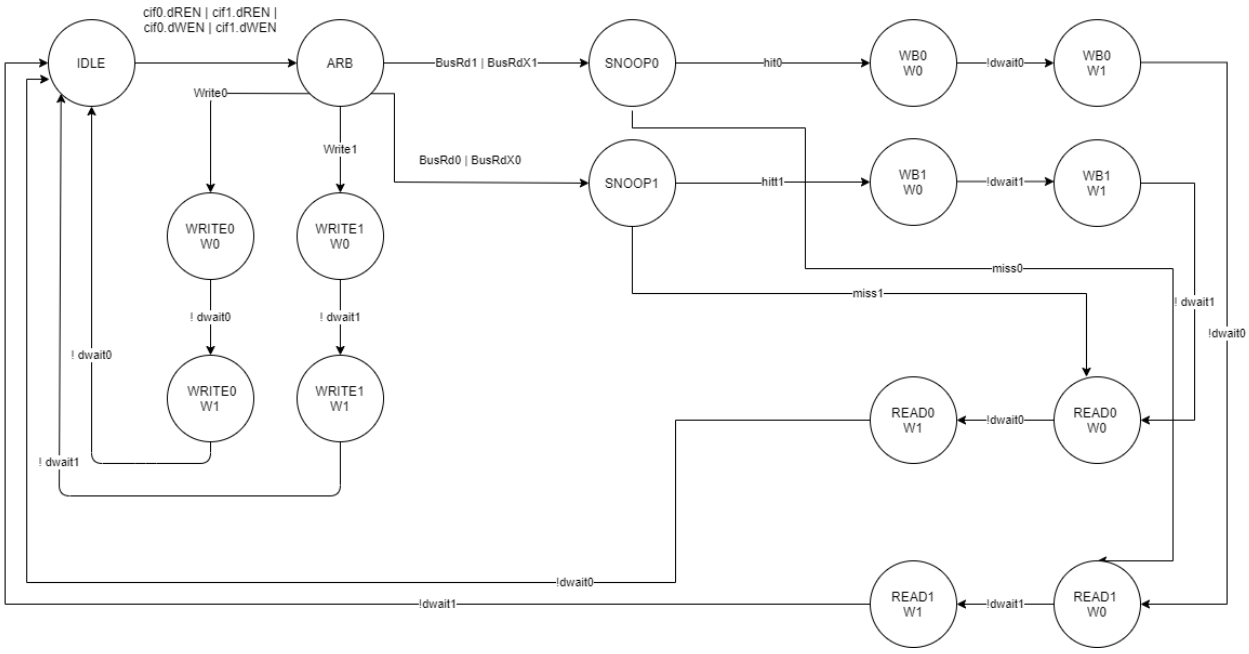


Figure 7: Bus Controller FSM

would be in the share state. While processor write would lead to the invalidation of the other cores frame if it was not already.

3 Results

Metric	Pipeline	Pipeline w/ Cache	Multicore
Max Frequency (MHz)	66.88	56.4	63.06
CPI ($^{\text{instr}}/_{\text{cycles}}$)	6.82	2.67	1.00
Instruction Latency ($^{\text{instr}}/_{\text{ns}}$)	272.99	106.81	40.19
Total Registers	1,678	4,150	9,208
Total Comb. Functions	3,428	7,065	17,878

Table 1: Comparison between single and dual core CPU's with a RAM latency of 5 cycles.

The results shown in Table 1 shows the data collected for the pipeline with and without

caches and the multicore processor design. The single core is tested with a single core merge sort algorithm and the multicore design is tested with a dual core shared memory merge sort algorithm. The maximum frequency was found in the log file produced during the synthesis of the processors RTL code. The merge sort program that the single core processors were tested with, when executed, consisted of 5,404 instructions. The merge sort program that the dual core processor were tested with, when executed, consisted of 5,421 instructions. The CPI for each of the processors was calculated using the formula $\text{CPI} = \frac{\#\text{CPU\_cycles}}{\#\text{Instructions}}$. Both programs were simulated using a gate-level simulation at 50 MHz. The instruction latency was calculated using the formula $\text{latency} = \frac{\text{total\_time}}{\#\text{Instructions}}$. The FPGA resources were found in the summary file produced during the synthesis of the mapped design to a representation of the mapped design on the lab FPGA's. Based on 1 and using the Iron Law, the net speedup from the single core pipeline with caches to the multicore can be calculated for the merge sort algorithm.

$$\text{time} = \frac{\text{instructions}}{\text{program}} \frac{\text{cycles}}{\text{instruction}} \frac{\text{seconds}}{\text{cycle}}$$

$$\text{speedup} = \frac{\text{time}_{\text{multicore}}}{\text{time}_{\text{singlecore}}}$$

$$\text{speedup} = \frac{5421 \cdot \frac{1}{1.00} \cdot \frac{1}{40.19}}{5404 \cdot \frac{1}{2.67} \cdot \frac{1}{108.81}} = 7.12$$

Metric	LAT: 0	LAT: 2	LAT: 4	LAT: 6	LAT: 8	LAT: 10
Single Core	1.28	2.55	3.83	5.11	6.39	7.66
Pipeline	2.42	3.7	5.5	6.8	8.1	9.37
Pipeline w/ Cache						

Table 2: The single cycle processor specifications.

The metric chosen to compare the processors across different memory latencies is CPI because it shows how well the caches are working at speeding up memory accesses.

4 Conclusion

The multicore design was intended to be more efficient than the pipeline design because the multicore design is made to create possible faster run time when utilizing two core. However, this is reliant as well on how effectively the program is written to utilize coherence. If the program is fairly inefficient then there would be minimal speed up while following a producer/consumer approach, you will notice a generous speedup.

Overall, we notice a great speedup in CPI and instruction latency from pipeline to pipeline with cache to multicore designs respectively. Specifically, we noticed a speedup of 7.12 from pipeline w/o cache to multicore implementation. Also there was a small decline in the max frequency, this is fairly negligible when taking account the total time at the respective max frequencies.

5 Contributions

Following the completion of our Pipeline Project, we kept a similar model where we generally worked on modules in parallel, while syncing up to confirm design diagrams as well as system implementation. As far as individual contributions, Chris designed the dcache, test bench for icache, test bench for bus controller, and full system implementation and debugging. Whereas, Victor designed memory and bus controller, icache, test bench for dcache, and created RTL and FSM diagrams for designs. As stated before we used most of our time in a team environment to discuss design and debug system level issues.